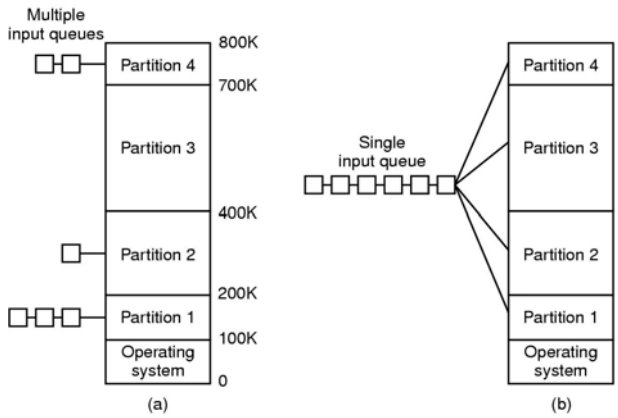
Memory Management

1 Basic memory management
2 Swapping
3 Virtual memory
4 Page replacement algorithms
5 Design issues for paging systems
6 Implementation issues
7 Segmentation

Memory Management

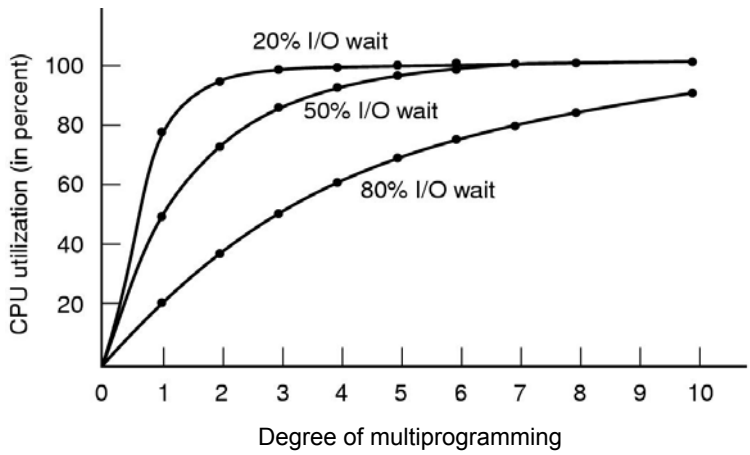
- Ideally programmers want memory that is
  - large
  - fast
  - non volatile
- Memory hierarchy
  - small amount of fast, expensive memory – cache
  - some medium-speed, medium price main memory
  - gigabytes of slow, cheap disk storage
- Memory manager handles the memory hierarchy

Multiprogramming with Fixed Partitions



- Fixed memory partitions
  - separate input queues for each partition
  - single input queue

Modeling Multiprogramming



CPU utilization as a function of number of processes in memory

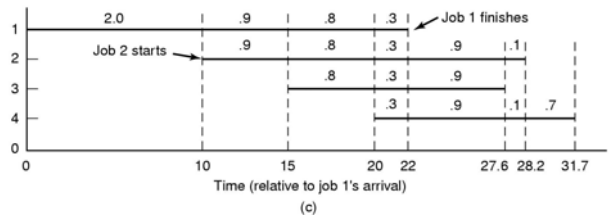
Analysis of Multiprogramming System Performance

Job	Arrival time	CPU minutes needed
1	10:00	4
2	10:10	3
3	10:15	2
4	10:20	2

(a)

	# Processes 1	2	3	4
CPU idle	.80	.64	.51	.41
CPU busy	.20	.36	.49	.59
CPU/process	.20	.18	.16	.15

(b)



(c)

- Arrival and work requirements of 4 jobs
- CPU utilization for 1 – 4 jobs with 80% I/O wait
- Sequence of events as jobs arrive and finish
  - note numbers show amount of CPU time jobs get in each interval

5

Multiprogramming Issues

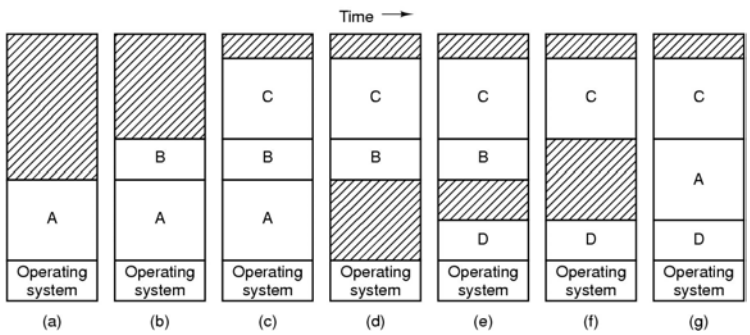
- Multiprogramming introduces the problems of relocation and protection
  - Cannot be sure where program will be loaded in memory
    - address locations of variables, code routines cannot be absolute
  - must keep a program out of other processes' partitions
- Solution: Use base and limit registers
  - Logical address added to base register to map to physical addr
  - address larger than limit value is an error

6

Memory Management for Timesharing Systems

- In batch systems, only running jobs need to be in memory which simplifies memory management
- In timesharing systems, there may not be enough physical memory for all active processes
- Two approaches
  - Swapping
  - Virtual memory
    - All modern systems support virtual memory

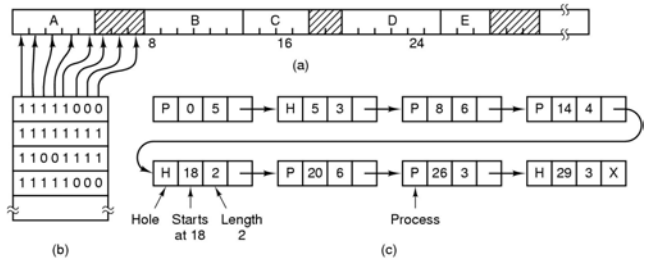
Swapping (1)



Memory allocation changes as
- processes come into memory
- leave memory

Shaded regions are unused memory

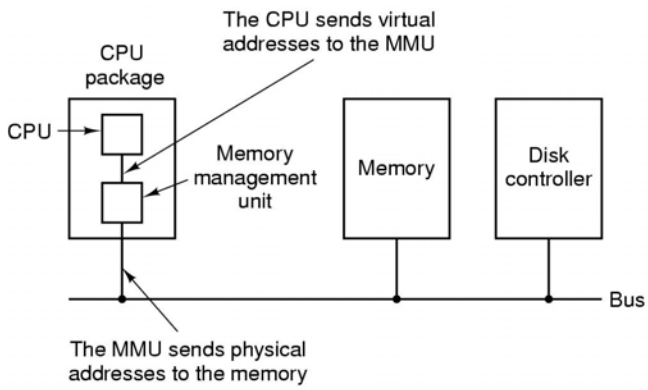
Memory Management with Bit Maps/Linked Lists



- Part of memory with 5 processes, 3 holes
  - tick marks show allocation units, shaded regions are free
  - OS can keep track of free regions via a bitmap or linked list
- Algorithms for memory allocation
  - First fit, Best fit, worst fit

Virtual Memory

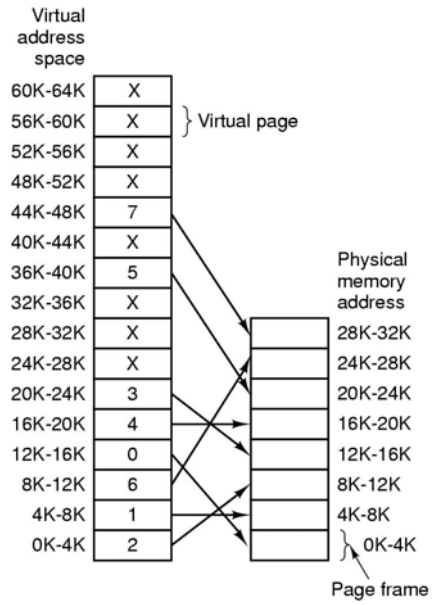
Paging (1)



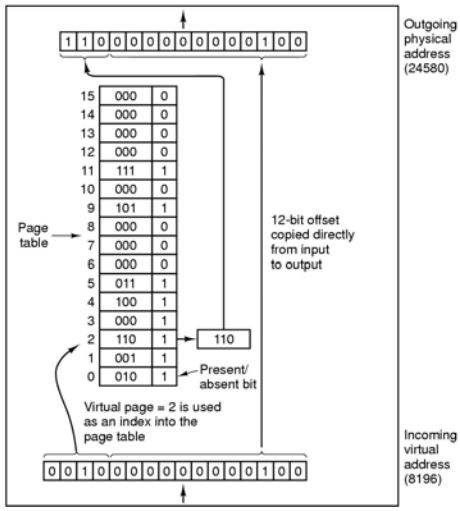
The position and function of the MMU

Paging (2)

The relation between virtual addresses and physical memory addresses given by page table

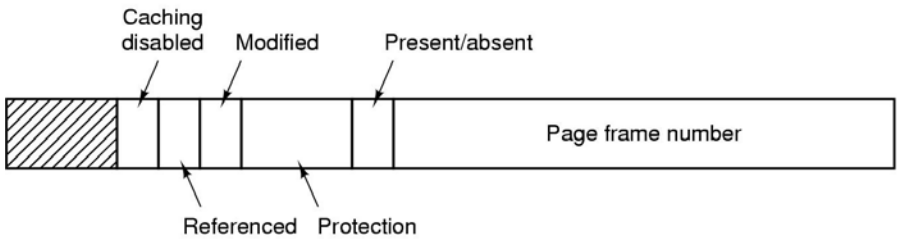


Page Tables (1)



Internal operation of MMU with 16 4 KB pages

Page Tables (2)

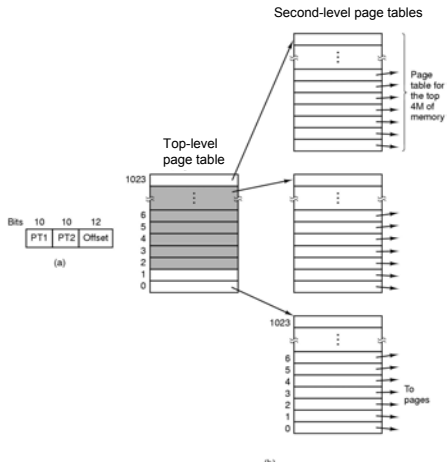


Typical page table entry

Page Tables (3)

- In modern computers, page tables can be very large
  - Virtual address = 32 bits, page size = 4 KB leads to 1 million page table entries
  - With 64 bit addresses, number of page table entries = 2^{52}
- Partial solution: multi-level page tables
  - Page the page tables, i.e., avoid keeping the entire page table in memory
- TLBs (Translation Lookaside Buffers)
  - Downside of multi-level paging: multiple levels of address translation needed for each memory access
  - Solution: use a special cache for speeding up address translation, I.e. cache recent virtual to physical page mappings
- Inverted Page Tables
  - Size depends upon physical memory
  - Always used in conjunction with TLBs

Page Tables (4)



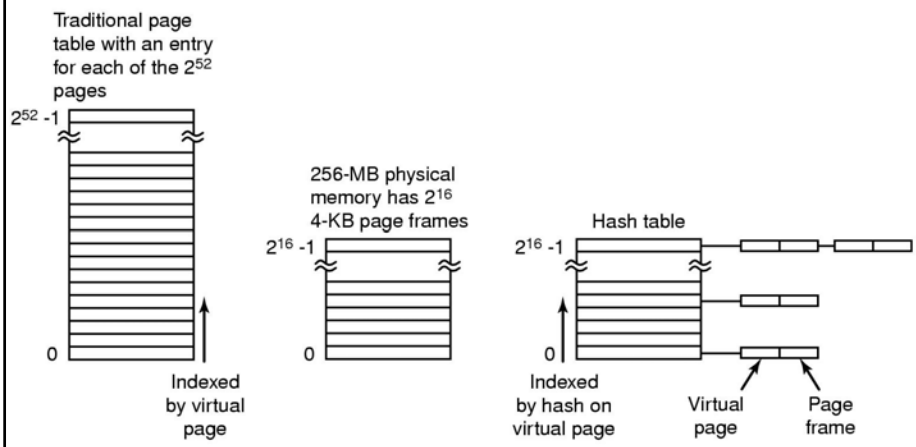
- 32 bit address with 2 page table fields
- Two-level page tables

TLBs – Translation Lookaside Buffers

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

A TLB to speed up paging

Inverted Page Tables



Comparison of a traditional page table with an inverted page table

Page Replacement Algorithms

- Page fault forces choice
  - which page must be removed
  - make room for incoming page
- Modified page must first be saved
  - unmodified just overwritten
- Better not to choose an often used page
  - will probably need to be brought back in soon

Optimal Page Replacement Algorithm

- Replace page needed at the farthest point in future
  - Optimal but unrealizable
- Estimate by …
  - logging page use on previous runs of process
  - although this is impractical

19

Not Recently Used Page Replacement Algorithm

- Each page has Reference bit, Modified bit
  - bits are set when page is referenced, modified
- Pages are classified
  1. not referenced, not modified
  2. not referenced, modified
  3. referenced, not modified
  4. referenced, modified
- NRU removes page at random
  - from lowest numbered non empty class

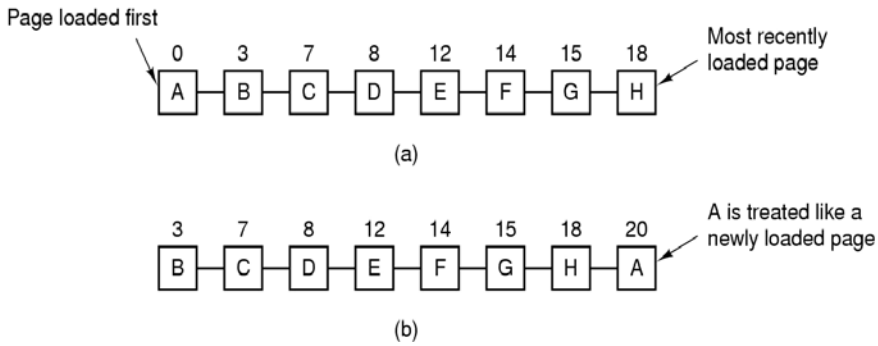
20

FIFO Page Replacement Algorithm

- Maintain a linked list of all pages
  - in order they came into memory
- Page at beginning of list replaced
- Disadvantage
  - page in memory the longest may be often used

21

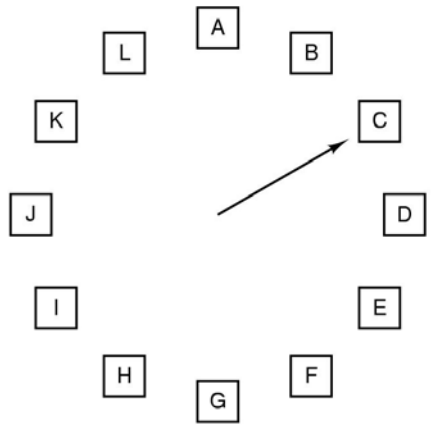
Second Chance Page Replacement Algorithm



- Operation of a second chance algorithm
  - pages sorted in FIFO order
  - Page list if fault occurs at time 20, *A* has *R* bit set (numbers above pages are loading times)

22

The Clock Page Replacement Algorithm



When a page fault occurs, the page the hand is pointing to is inspected. The action taken depends on the R bit:

R = 0: Evict the page

R = 1: Clear R and advance hand

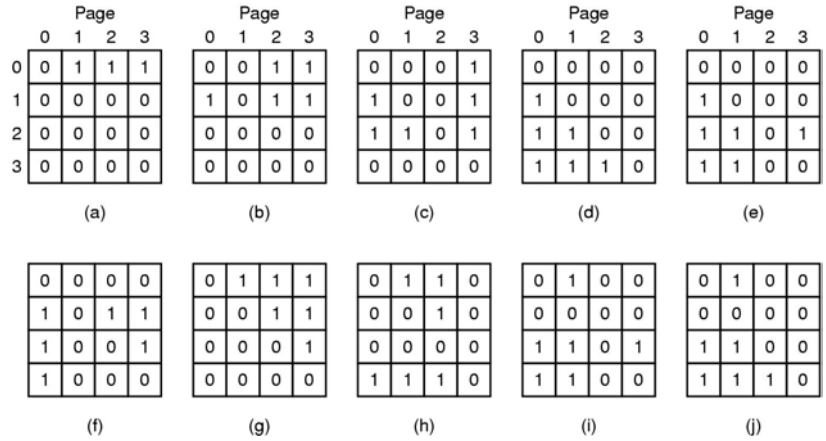
23

Least Recently Used (LRU)

- Assume pages used recently will used again soon
  - throw out page that has been unused for longest time
- Must keep a linked list of pages
  - most recently used at front, least at rear
  - update this list every memory reference !!
- Alternatively keep counter in each page table entry
  - choose page with lowest value counter
  - periodically zero the counter

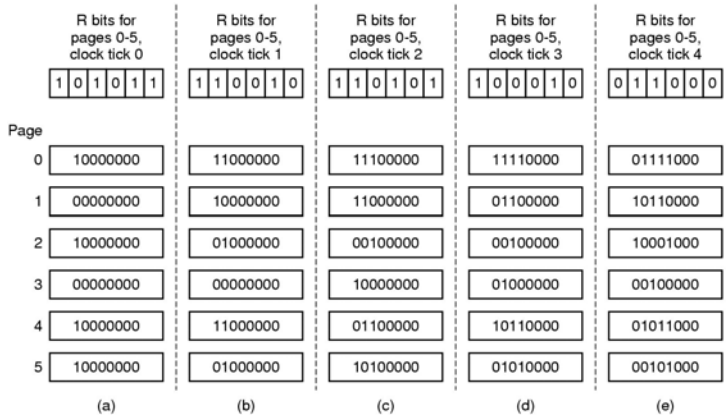
24

Simulating LRU in Software (1)



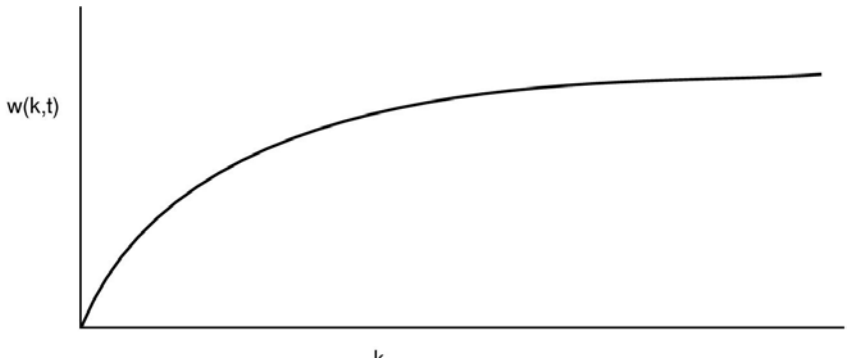
LRU using a matrix – pages referenced in order
0,1,2,3,2,1,0,3,2,3

Simulating LRU in Software (2)



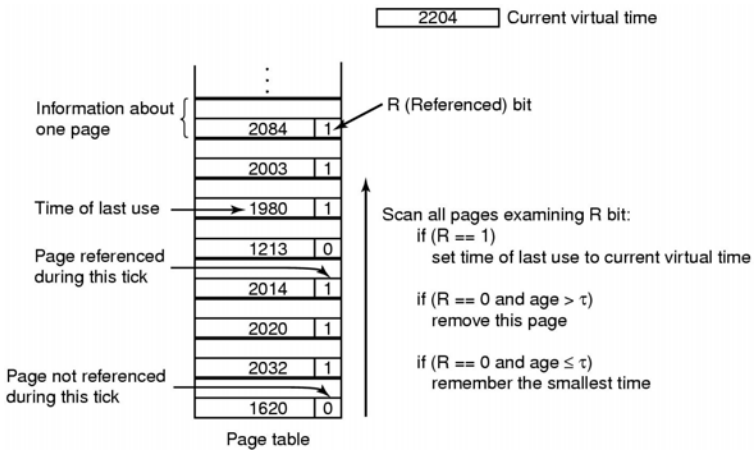
- The aging algorithm simulates LRU in software
- Note 6 pages for 5 clock ticks, (a) – (e)

The Working Set Page Replacement Algorithm (1)



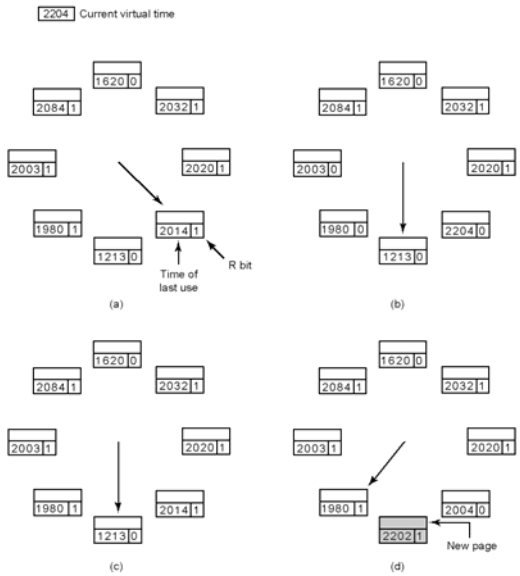
- The working set is the set of pages used by the *k* most recent memory references
- w(k,t) is the size of the working set at time, *t*

The Working Set Page Replacement Algorithm (2)



The working set algorithm

The WSClock Page Replacement Algorithm



Review of Page Replacement Algorithms

Algorithm	Comment
Optimal	Not implementable, but useful as a benchmark
NRU (Not Recently Used)	Very crude
FIFO (First-In, First-Out)	Might throw out important pages
Second chance	Big improvement over FIFO
Clock	Realistic
LRU (Least Recently Used)	Excellent, but difficult to implement exactly
NFU (Not Frequently Used)	Fairly crude approximation to LRU
Aging	Efficient algorithm that approximates LRU well
Working set	Somewhat expensive to implement
WSClock	Good efficient algorithm

Modeling Page Replacement Algorithms
Belady's Anomaly

All pages frames initially empty

		0	1	2	3	0	1	4	0	1	2	3	4
Youngest page		0	1	2	3	0	1	4	4	4	2	3	3
			0	1	2	3	0	1	1	1	4	2	2
Oldest page				0	1	2	3	0	0	0	1	4	4
		P	P	P	P	P	P	P			P	P	

9 Page faults

(a)

		0	1	2	3	0	1	4	0	1	2	3	4
Youngest page		0	1	2	3	3	3	4	0	1	2	3	4
			0	1	2	2	2	3	4	0	1	2	3
Oldest page				0	1	1	1	2	3	4	0	1	2
					0	0	0	1	2	3	4	0	1
		P	P	P	P			P	P	P	P	P	P

10 Page faults

(b)

- FIFO with 3 page frames
- FIFO with 4 page frames
- *P*'s show which page references show page faults

Stack Algorithms

Reference string	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	1	3	4	1
	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	1	3	4	1
		0	2	1	3	5	4	6	3	7	4	7	7	3	3	5	3	3	3	1	7	1	3	4
			0	2	1	3	5	4	6	3	3	4	4	7	7	7	5	5	5	3	3	7	1	3
				0	2	1	3	5	4	6	6	6	6	4	4	4	7	7	7	5	5	5	7	7
					0	2	1	1	5	5	5	5	5	6	6	6	4	4	4	4	4	4	5	5
						0	2	2	1	1	1	1	1	1	1	1	6	6	6	6	6	6	6	6
							0	0	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
									0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Page faults	P	P	P	P	P	P	P		P					P			P						P	
Distance string	∞	∞	∞	∞	∞	∞	∞	4	∞	4	2	3	1	5	1	2	6	1	1	4	2	3	5	3

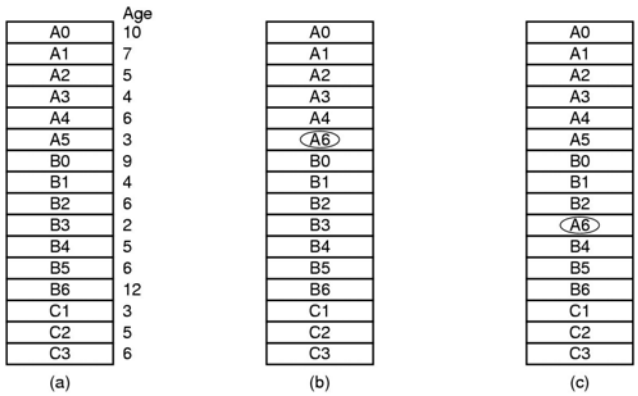
7 4 6 5

State of memory array, *M*, after each item in reference string is processed

Stack algorithms like LRU do not exhibit Belady's anomaly

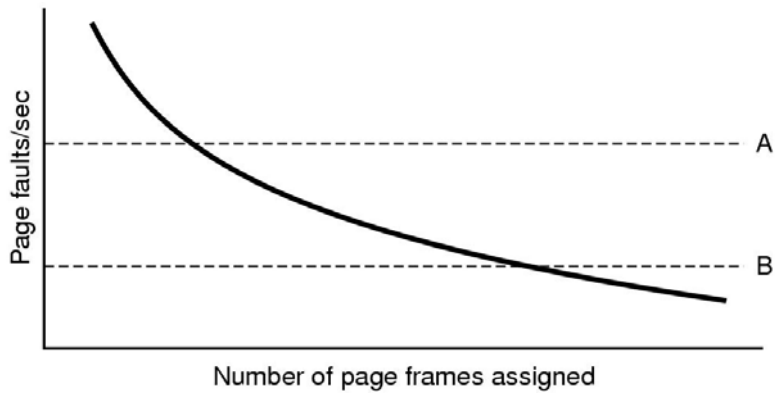
Design Issues for Paging Systems

Local versus Global Allocation Policies (1)



- Original configuration
- Local page replacement
- Global page replacement

Local versus Global Allocation Policies (2)



Page fault rate as a function of the number of page frames assigned

Load Control

- Despite good designs, system may still thrash
- When PFF algorithm indicates
  - some processes need more memory
  - but no processes need less
- Solution :
  Reduce number of processes competing for memory
  - swap one or more to disk, divide up pages they held
  - reconsider degree of multiprogramming

35

Page Size (1)

Small page size

- Advantages
  - less internal fragmentation
  - better fit for various data structures, code sections
  - less unused program in memory
- Disadvantages
  - programs need many pages, larger page tables

36

Page Size (2)

- Overhead due to page table and internal fragmentation

$$overhead = \frac{s \cdot e}{p} + \frac{p}{2}$$

page table space

internal fragmentation

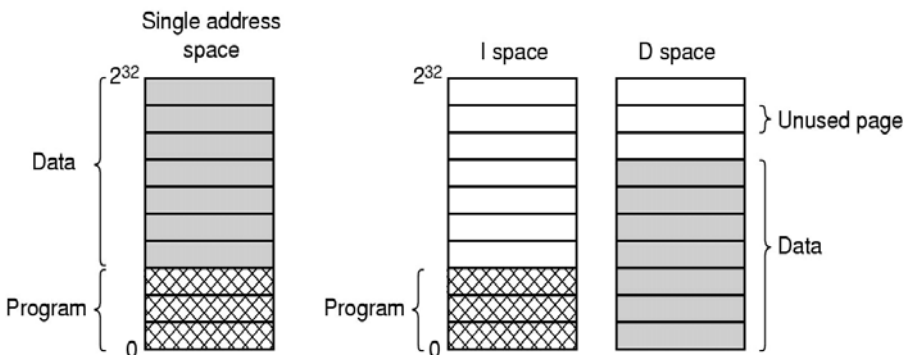
- Where
  - s = average process size in bytes
  - p = page size in bytes
  - e = page entry

Optimized when

$$p = \sqrt{2se}$$

37

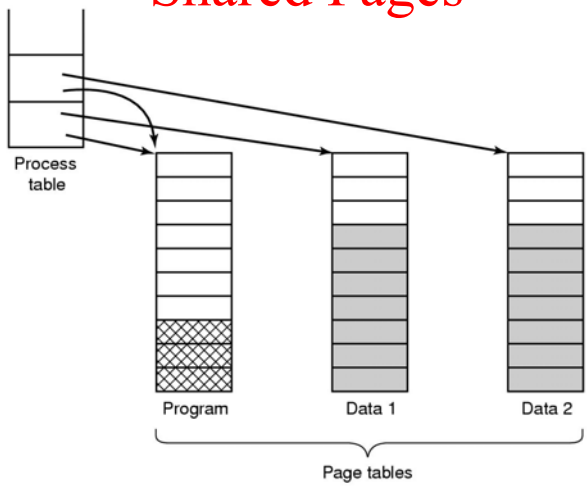
Separate Instruction and Data Spaces



- One address space
- Separate I and D spaces

38

Shared Pages



Two processes sharing same program sharing its page table

39

Cleaning Policy

- Need for a background process, paging daemon
  - periodically inspects state of memory
- When too few frames are free
  - selects pages to evict using a replacement algorithm
- It can use same circular list (clock)
  - as regular page replacement algorithm but with different pointer

40

Implementation Issues

Operating System Involvement with Paging

Four times when OS involved with paging

1. Process creation
   - determine program size
   - create page table
2. Process execution
   - MMU reset for new process
   - TLB flushed
3. Page fault time
   - determine virtual address causing fault
   - swap target page out, needed page in
4. Process termination time
   - release page table, pages

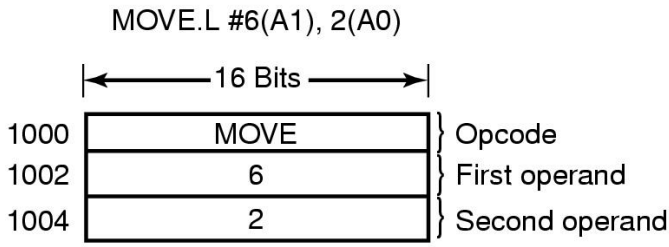
41

Page Fault Handling

1. Hardware traps to kernel
2. General registers saved
3. OS determines which virtual page needed
4. OS checks validity of address, seeks page frame
5. If selected frame is dirty, write it to disk
6. OS brings schedules new page in from disk
7. Page tables updated
8. Faulting instruction backed up to when it began
9. Faulting process scheduled
10. Registers restored
11. Program continues

42

Instruction Backup



An instruction causing a page fault

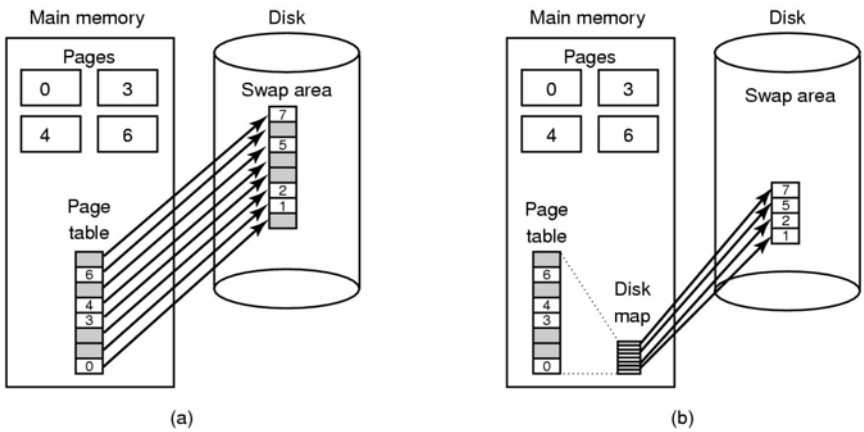
43

Locking Pages in Memory

- Virtual memory and I/O occasionally interact
- Proc issues call for read from device into buffer
  - while waiting for I/O, another processes starts up
  - has a page fault
  - buffer for the first proc may be chosen to be paged out
- Need to specify some pages locked
  - exempted from being target pages

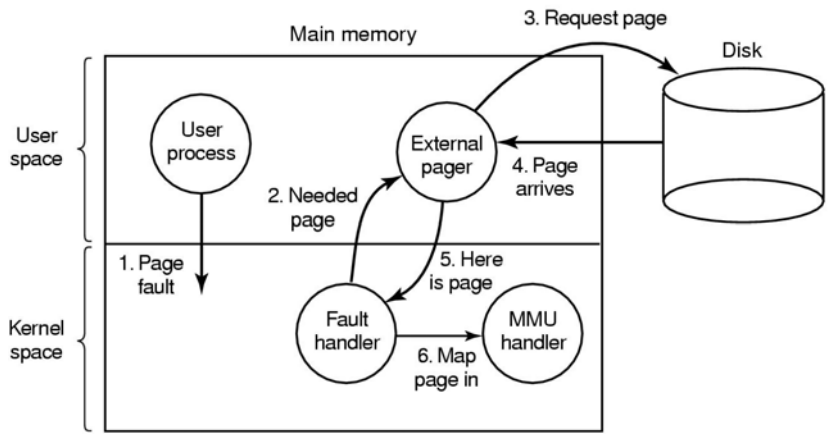
44

Backing Store



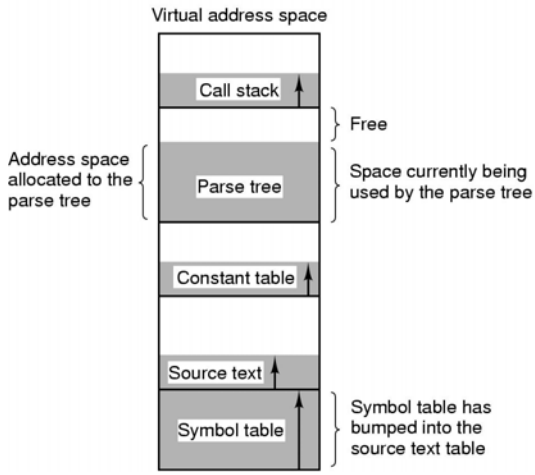
(a) Paging to static swap area
(b) Backing up pages dynamically

Separation of Policy and Mechanism



Page fault handling with an external pager

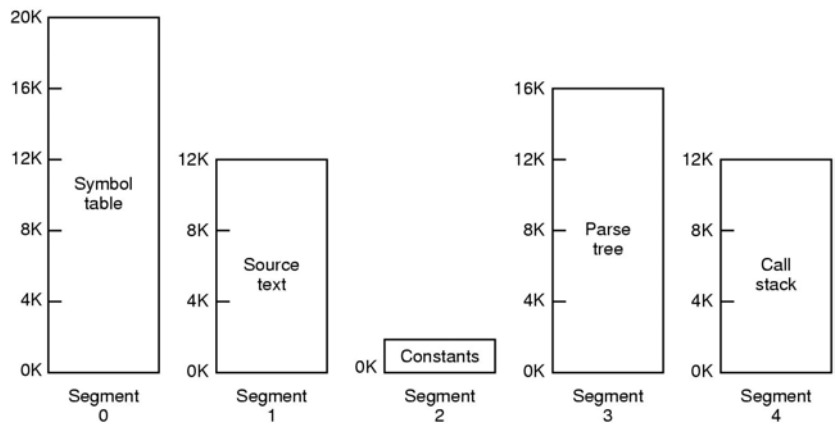
Segmentation (1)



- One-dimensional address space with growing tables
- One table may bump into another

47

Segmentation (2)



Allows each table to grow or shrink, independently

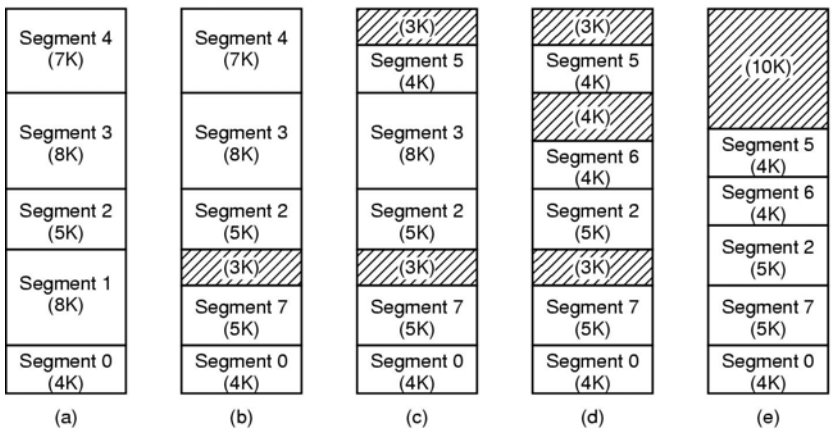
48

Segmentation (3)

Consideration	Paging	Segmentation
Need the programmer be aware that this technique is being used?	No	Yes
How many linear address spaces are there?	1	Many
Can the total address space exceed the size of physical memory?	Yes	Yes
Can procedures and data be distinguished and separately protected?	No	Yes
Can tables whose size fluctuates be accommodated easily?	No	Yes
Is sharing of procedures between users facilitated?	No	Yes
Why was this technique invented?	To get a large linear address space without having to buy more physical memory	To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection

Comparison of paging and segmentation

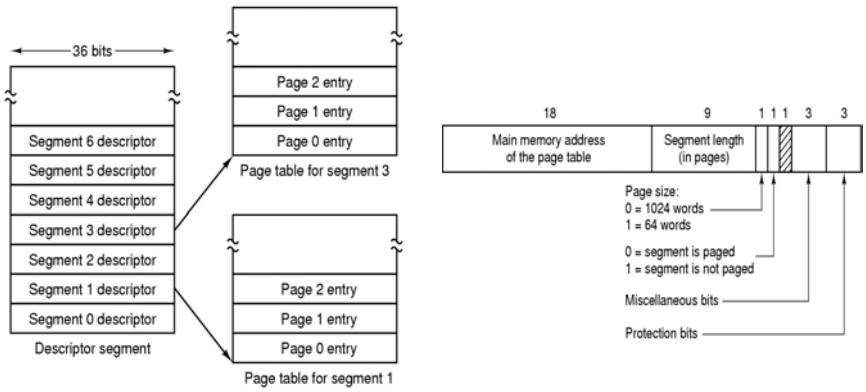
Implementation of Pure Segmentation



(a)-(d) Development of checkerboarding

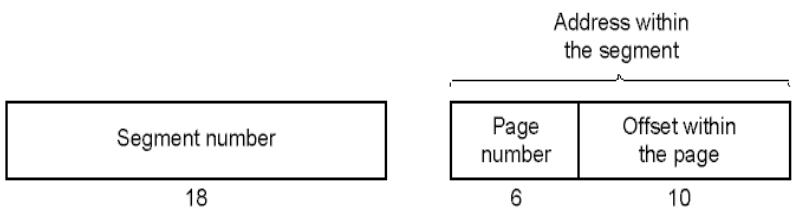
(e) Removal of the checkerboarding by compaction

Segmentation with Paging: MULTICS (1)



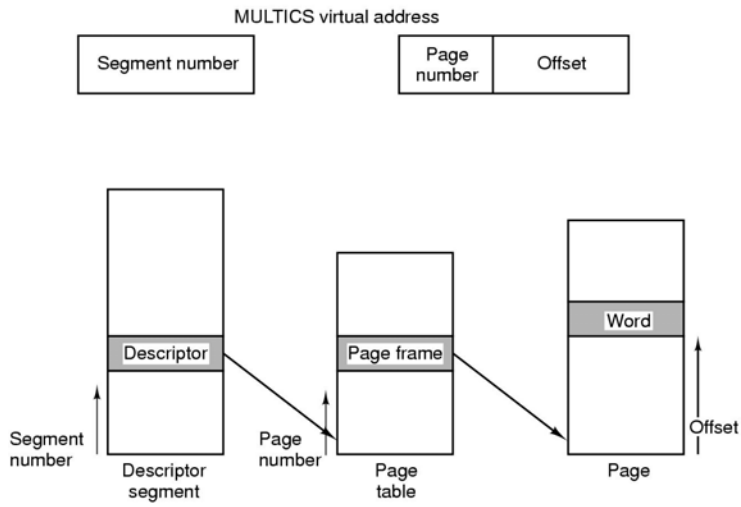
- Descriptor segment points to page tables
- Segment descriptor – numbers are field lengths

Segmentation with Paging: MULTICS (2)



A 34-bit MULTICS virtual address

Segmentation with Paging: MULTICS (3)



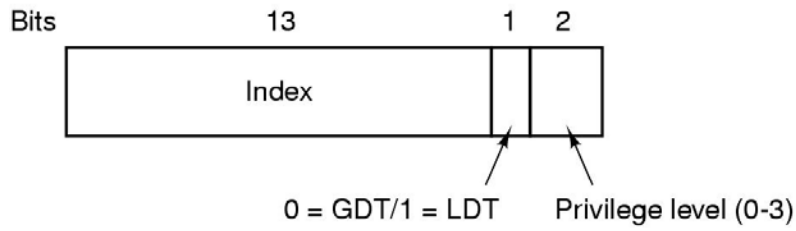
Conversion of a 2-part MULTICS address into a main memory address

Segmentation with Paging: MULTICS (4)

Comparison field: Segment number	Comparison field: Virtual page	Page frame	Protection	Age	Is this entry used?
4	1	7	Read/write	13	1
6	0	2	Read only	10	1
12	3	1	Read/write	2	1
					0
2	1	0	Execute only	7	1
2	2	12	Execute only	9	1

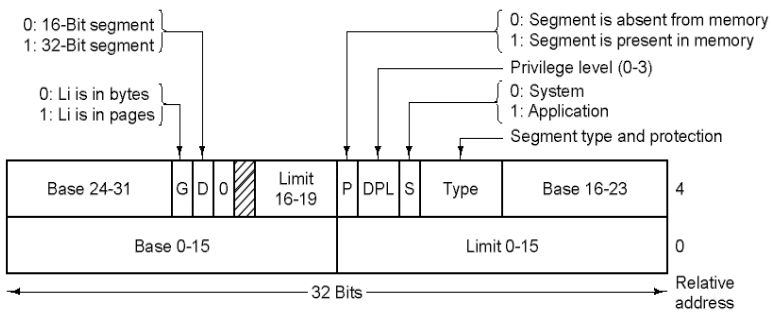
- Simplified version of the MULTICS TLB
- Existence of 2 page sizes makes actual TLB more complicated

Segmentation with Paging: Pentium (1)



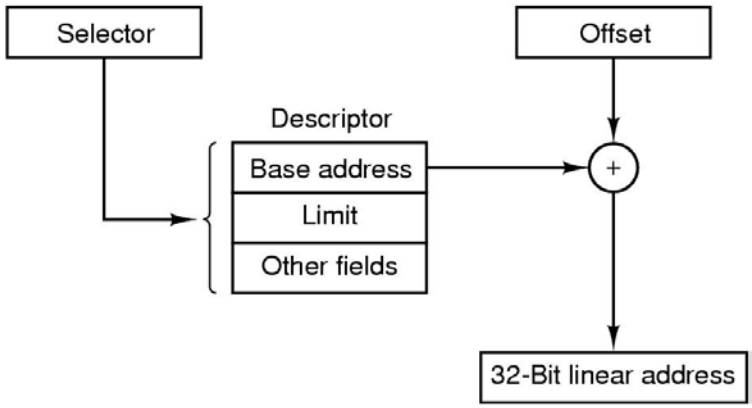
A Pentium selector

Segmentation with Paging: Pentium (2)



- Pentium code segment descriptor
- Data segments differ slightly

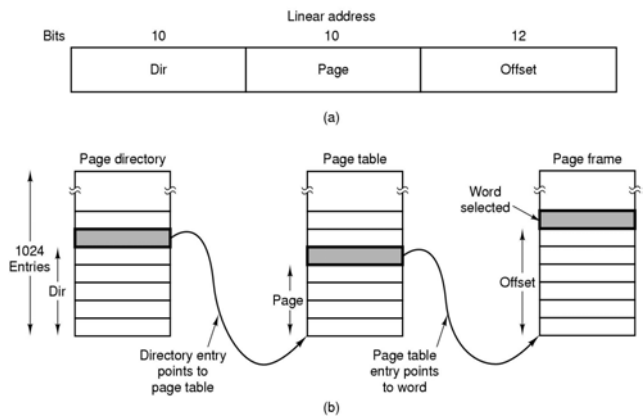
Segmentation with Paging: Pentium (3)



Conversion of a (selector, offset) pair to a linear address

57

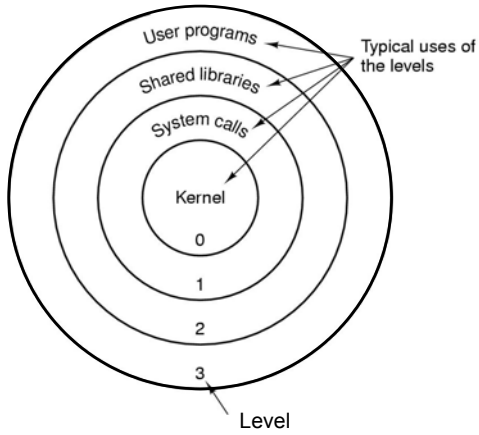
Segmentation with Paging: Pentium (4)



Mapping of a linear address onto a physical address

58

Segmentation with Paging: Pentium (5)



Protection on the Pentium

59